Tracking Immediate Predecessors in Distributed Computations

Emmanuelle Anceaume Jean-Michel Hélary Michel Raynal
IRISA, Campus Beaulieu
35042 Rennes Cedex, France
FirstName.LastName@irisa.fr

ABSTRACT

A distributed computation is usually modeled as a partially ordered set of relevant events (the relevant events are a subset of the primitive events produced by the computation). An important causality-related distributed computing problem, that we call the *Immediate Predecessors Tracking* (IPT) problem, consists in associating with each relevant event, on the fly and without using additional control messages, the set of relevant events that are its immediate predecessors in the partial order. So, IPT is the on-the-fly computation of the transitive reduction (i.e., Hasse diagram) of the causality relation defined by a distributed computation. This paper addresses the IPT problem: it presents a family of protocols that provides each relevant event with a timestamp that exactly identifies its immediate predecessors. The family is defined by a general condition that allows application messages to piggyback control information whose size can be smaller than n (the number of processes). In that sense, this family defines message size-efficient IPT protocols. According to the way the general condition is implemented, different IPT protocols can be obtained. Two of them are exhibited.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]:

General Terms

Asynchronous Distributed Computations

Keywords

Causality Tracking, Hasse Diagram, Immediate Predecessor, Message-Passing, Timestamp, Vector Clock.

1. INTRODUCTION

A distributed computation consists of a set of processes that cooperate to achieve a common goal. A main characteristic of these computations lies in the fact that the processes do not share a common global memory, and communicate only by exchanging messages over a communication network. Moreover, message transfer delays are finite but unpredictable. This computation model defines what is known as the *asynchronous distributed system model*. It is particularly important as it includes systems that span large geographic areas, and systems that are subject to unpredictable loads. Consequently, the concepts, tools and mechanisms developed for asynchronous distributed systems reveal to be both important and general.

Causality is a key concept to understand and master the behavior of asynchronous distributed systems [18]. More precisely, given two events e and f of a distributed computation, a crucial problem that has to be solved in a lot of distributed applications is to know whether they are causally related, i.e., if the occurrence of one of them is a consequence of the occurrence of the other. The causal past of an event e is the set of events from which e is causally dependent. Events that are not causally dependent are said to be concurrent. Vector clocks [5, 16] have been introduced to allow processes to track causality (and concurrency) between the events they produce. The timestamp of an event produced by a process is the current value of the vector clock of the corresponding process. In that way, by associating vector timestamps with events it becomes possible to safely decide whether two events are causally related or not.

Usually, according to the problem he focuses on, a designer is interested only in a subset of the events produced by a distributed execution (e.g., only the checkpoint events are meaningful when one is interested in determining consistent global checkpoints [12]). It follows that detecting causal dependencies (or concurrency) on all the events of the distributed computation is not desirable in all applications [7, 15]. In other words, among all the events that may occur in a distributed computation, only a subset of them are relevant. In this paper, we are interested in the restriction of the causality relation to the subset of events defined as being the relevant events of the computation.

Being a strict partial order, the causality relation is transitive. As a consequence, among all the relevant events that causally precede a given relevant event e, only a subset are its immediate predecessors: those are the events f such that there is no relevant event on any causal path from f to e. Unfortunately, given only the vector timestamp associated with an event it is not possible to determine which events of its causal past are its immediate predecessors. This comes from the fact that the vector timestamp associated with e determines, for each process, the last relevant event belong-

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.

ing to the causal past of e, but such an event is not necessarily an immediate predecessor of e. However, some applications [4, 6] require to associate with each relevant event only the set of its immediate predecessors. Those applications are mainly related to the analysis of distributed computations. Some of those analyses require the construction of the lattice of consistent cuts produced by the computation [15, 16]. It is shown in [4] that the tracking of immediate predecessors allows an efficient on the fly construction of this lattice. More generally, these applications are interested in the very structure of the causal past. In this context, the determination of the immediate predecessors becomes a major issue [6]. Additionally, in some circumstances, this determination has to satisfy behavior constraints. If the communication pattern of the distributed computation cannot be modified, the determination has to be done without adding control messages. When the immediate predecessors are used to monitor the computation, it has to be done on the fly.

We call *Immediate Predecessor Tracking* (IPT) the problem that consists in determining on the fly and without additional messages the immediate predecessors of relevant events. This problem consists actually in determining the transitive reduction (Hasse diagram) of the causality graph generated by the relevant events of the computation. Solving this problem requires tracking causality, hence using vector clocks. Previous works have addressed the efficient implementation of vector clocks to track causal dependence on relevant events. Their aim was to reduce the size of timestamps attached to messages. An efficient vector clock implementation suited to systems with FIFO channels is proposed in [19]. Another efficient implementation that does not depend on channel ordering property is described in [11]. The notion of *causal barrier* is introduced in [2, 17] to reduce the size of control information required to implement causal multicast. However, none of these papers considers the IPT problem. This problem has been addressed for the first time (to our knowledge) in [4, 6] where an IPT protocol is described, but without correctness proof. Moreover, in this protocol, timestamps attached to messages are of size n. This raises the following question which, to our knowledge, has never been answered: "Are there efficient vector clock implementation techniques that are suitable for the IPT problem?".

This paper has three main contributions: (1) a positive answer to the previous open question, (2) the design of a family of efficient IPT protocols, and (3) a formal correctness proof of the associated protocols. From a methodological point of view the paper uses a top-down approach. It states abstract properties from which more concrete properties and protocols are derived. The family of IPT protocols is defined by a general condition that allows application messages to piggyback control information whose size can be smaller than the system size (i.e., smaller than the number of processes composing the system). In that sense, this family defines low cost IPT protocols when we consider the message size. In addition to efficiency, the proposed approach has an interesting design property. Namely, the family is incrementally built in three steps. The basic vector clock protocol is first enriched by adding to each process a boolean vector whose management allows the processes to track the immediate predecessor events. Then, a general condition is stated to reduce the size of the control information carried by messages. Finally, according to the way this condition is implemented, three IPT protocols are obtained.

The paper is composed of seven sections. Sections 2 introduces the computation model, vector clocks and the notion of relevant events. Section 3 presents the first step of the construction that results in an IPT protocol in which each message carries a vector clock and a boolean array, both of size n (the number of processes). Section 4 improves this protocol by providing the general condition that allows a message to carry control information whose size can be smaller than n. Section 5 provides instantiations of this condition. Section 6 provides a simulation study comparing the behaviors of the proposed protocols. Finally, Section 7 concludes the paper. (Due to space limitations, proofs of lemmas and theorems are omitted. They can be found in [1].)

2. MODEL AND VECTOR CLOCK

2.1 Distributed Computation

A distributed program is made up of sequential local programs which communicate and synchronize only by exchanging messages. A distributed computation describes the execution of a distributed program. The execution of a local program gives rise to a sequential process. Let $\{P_1, P_2, \ldots, P_n\}$ be the finite set of sequential processes of the distributed computation. Each ordered pair of communicating processes (P_i, P_j) is connected by a reliable channel c_{ij} through which P_i can send messages to P_j. We assume that each message is unique and a process does not send messages to itself[1]. Message transmission delays are finite but unpredictable. Moreover, channels are not necessarily FIFO. Process speeds are positive but arbitrary. In other words, the underlying computation model is asynchronous.

The local program associated with P_i can include send, receive and internal statements. The execution of such a statement produces a corresponding send/receive/internal event. These events are called *primitive events*. Let e_i^x be the x-th event produced by process P_i. The sequence $h_i = e_i^1 e_i^2 \ldots e_i^x \ldots$ constitutes the history of P_i, denoted H_i. Let $H = \cup_{i=1}^{n} H_i$ be the set of events produced by a distributed computation. This set is structured as a partial order by Lamport's *happened before* relation [14] (denoted $\stackrel{hb}{\rightarrow}$) and defined as follows: $e_i^x \stackrel{hb}{\rightarrow} e_j^y$ if and only if

$(i = j \wedge x + 1 = y)$ (local precedence) $\vee$
$(\exists m : e_i^x = send(m) \wedge e_j^y = receive(m))$ (msg prec.) $\vee$
$(\exists\, e_k^z : e_i^x \stackrel{hb}{\rightarrow} e_k^z \wedge e_k^z \stackrel{hb}{\rightarrow} e_j^y)$ (transitive closure).

$\max(e_i^x, e_j^y)$ is a partial function defined only when e_i^x and e_j^y are ordered. It is defined as follows: $\max(e_i^x, e_j^y) = e_i^x$ if $e_j^y \stackrel{hb}{\rightarrow} e_i^x$, $\max(e_i^x, e_j^y) = e_i^y$ if $e_i^x \stackrel{hb}{\rightarrow} e_j^y$.

Clearly the restriction of $\stackrel{hb}{\rightarrow}$ to H_i, for a given i, is a total order. Thus we will use the notation $e_i^x < e_i^y$ iff $x < y$. Throughout the paper, we will use the following notation: if $e \in H_i$ is not the first event produced by P_i, then $pred(e)$ denotes the event immediately preceding e in the sequence H_i. If e is the first event produced by P_i, then $pred(e)$ is denoted by $\perp$ (meaning that there is no such event), and $\forall e \in H_i : \perp < e$. The partial order $\widehat{H} = (H, \stackrel{hb}{\rightarrow})$ constitutes a formal model of the distributed computation it is associated with.

[1]This assumption is only in order to get simple protocols.

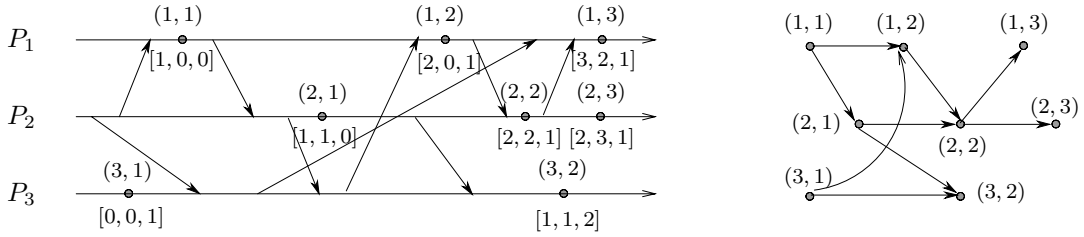


Figure 1: Timestamped Relevant Events and Immediate Predecessors Graph (Hasse Diagram)

2.2 Relevant Events

For a given *observer* of a distributed computation, only some events are relevant[2] [7, 9, 15]. An interesting example of "what an observation is", is the detection of predicates on consistent global states of a distributed computation [3, 6, 8, 9, 13, 15]. In that case, a relevant event corresponds to the modification of a local variable involved in the global predicate. Another example is the checkpointing problem where a relevant event is the definition of a local checkpoint [10, 12, 20].

The left part of Figure 1 depicts a distributed computation using the classical space-time diagram. In this figure, only relevant events are represented. The sequence of relevant events produced by process P_i is denoted by R_i, and $R = \cup_{i=1}^{n} R_i \subseteq H$ denotes the set of all relevant events. Let $\rightarrow$ be the relation on R defined in the following way:

$$\forall\ (e, f) \in R \times R : (e \rightarrow f) \Leftrightarrow (e \stackrel{hb}{\rightarrow} f).$$

The poset $(R, \rightarrow)$ constitutes an abstraction of the distributed computation [7]. In the following we consider a distributed computation at such an abstraction level. Moreover, without loss of generality we consider that the set of relevant events is a subset of the internal events (if a communication event has to be observed, a relevant internal event can be generated just before a *send* and just after a *receive* communication event occurred). Each relevant event is identified by a pair (process id, sequence number) (see Figure 1).

DEFINITION 1. *The* relevant causal past *of an event $e \in H$ is the (partially ordered) subset of* relevant *events f such that $f \stackrel{hb}{\rightarrow} e$. It is denoted $\uparrow(e)$. We have $\uparrow(e) = \{f \in R \mid f \stackrel{hb}{\rightarrow} e\}$.*

Note that, if $e \in R$ then $\uparrow(e) = \{f \in R \mid f \rightarrow e\}$. In the computation described in Figure 1, we have, for the event e identified $(2, 2)$: $\uparrow(e) = \{(1, 1), (1, 2), (2, 1), (3, 1)\}$. The following properties are immediate consequences of the previous definitions. Let $e \in H$.

CP1 If e is not a receive event then

$$\uparrow(e) = \begin{cases} \emptyset \text{ if } pred(e) = \perp, \\ \uparrow(pred(e)) \cup \{pred(e)\} \text{ if } pred(e) \in R, \\ \uparrow(pred(e)) \text{ if } pred(e) \notin R. \end{cases}$$

CP2 If e is a receive event (of a message m) then

$$\uparrow(e) = \begin{cases} \uparrow(send(m)) \text{ if } pred(e) = \perp, \\ \uparrow(pred(e)) \cup \uparrow(send(m)) \cup \{pred(e)\} \\ \qquad \text{if } pred(e) \in R, \\ \uparrow(pred(e)) \cup \uparrow(send(m)) \text{ if } pred(e) \notin R. \end{cases}$$

[2]Those events are sometimes called *observable* events.

DEFINITION 2. *Let $e \in H_i$. For every j such that $\uparrow(e) \cap R_j \neq \emptyset$, the* last relevant event *of P_j with respect to e is: $lastr(e, j) = \max\{f \mid f \in \uparrow(e) \cap R_j\}$. When $\uparrow(e) \cap R_j = \emptyset$, $lastr(e, j)$ is denoted by $\perp$ (meaning that there is no such event).*

Let us consider the event e identified (2,2) in Figure 1. We have $lastr(e, 1) = (1, 2)$, $lastr(e, 2) = (2, 1)$, $lastr(e, 3) = (3, 1)$. The following properties relate the events $lastr(e, j)$ and $lastr(f, j)$ for all the predecessors f of e in the relation $\stackrel{hb}{\rightarrow}$. These properties follow directly from the definitions. Let $e \in H_i$.

LR0 $\forall e \in H_i$:

$$lastr(e, i) = \begin{cases} \perp \text{ if } pred(e) = \perp, \\ pred(e) \text{ if } pred(e) \in R, \\ lastr(pred(e), i) \text{ if } pred(e) \notin R. \end{cases}$$

LR1 If e is not a receipt event: $\forall j \neq i$:
$lastr(e, j) = lastr(pred(e), j)$.

LR2 If e is a receive event of m: $\forall j \neq i$:
$lastr(e, j) = \max(lastr(pred(e), j), lastr(send(m), j))$.

2.3 Vector Clock System

Definition As a fundamental concept associated with the causality theory, vector clocks have been introduced in 1988, simultaneously and independently by Fidge [5] and Mattern [16]. A vector clock system is a mechanism that associates timestamps with events in such a way that the comparison of their timestamps indicates whether the corresponding events are or are not causally related (and, if they are, which one is the first). More precisely, each process P_i has a vector of integers $VC_i[1..n]$ such that $VC_i[j]$ is the number of relevant events produced by P_j, that belong to the current relevant causal past of P_i. Note that $VC_i[i]$ counts the number of relevant events produced so far by P_i. When a process P_i produces a (relevant) event e, it associates with e a vector timestamp whose value (denoted $e.VC$) is equal to the current value of VC_i.

Vector Clock Implementation The following implementation of vector clocks [5, 16] is based on the observation that $\forall i,\ \forall e \in H_i,\ \forall j\ :\ e.VC_i[j] = y \Leftrightarrow lastr(e, j) = e_j^y$ where $e.VC_i$ is the value of VC_i just after the occurrence of e (this relation results directly from the properties LR0, LR1, and LR2). Each process P_i manages its vector clock $VC_i[1..n]$ according to the following rules:

VC0 $VC_i[1..n]$ is initialized to $[0, \ldots, 0]$.

VC1 Each time it produces a relevant event e, P_i increments its vector clock entry $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$) to

indicate it has produced one more relevant event, then P_i associates with e the timestamp $e.VC = VC_i$.

VC2 When a process P_i sends a message m, it attaches to m the current value of VC_i. Let $m.VC$ denote this value.

VC3 When P_i receives a message m, it updates its vector clock as follows: $\forall k : VC_i[k] := max(VC_i[k], m.VC[k])$.

3. IMMEDIATE PREDECESSORS

In this section, the *Immediate Predecessor Tracking* (IPT) problem is stated (Section 3.1). Then, some technical properties of immediate predecessors are stated and proved (Section 3.2). These properties are used to design the basic IPT protocol and prove its correctness (Section 3.3). This IPT protocol, previously presented in [4] without proof, is built from a vector clock protocol by adding the management of a local boolean array at each process.

3.1 The IPT Problem

As indicated in the introduction, some applications (e.g., analysis of distributed executions [6], detection of distributed properties [7]) require to determine (on-the-fly and without additional messages) the transitive reduction of the relation $\rightarrow$ (i.e., we must not consider transitive causal dependency). Given two relevant events f and e, we say that f is an immediate predecessor of e if $f \rightarrow e$ and there is no relevant event g such that $f \rightarrow g \rightarrow e$.

DEFINITION 3. *The Immediate Predecessor Tracking (IPT) problem consists in associating with each relevant event e the set of relevant events that are its immediate predecessors. Moreover, this has to be done on the fly and without additional control message (i.e., without modifying the communication pattern of the computation).*

As noted in the Introduction, the IPT problem is the computation of the Hasse diagram associated with the partially ordered set of the relevant events produced by a distributed computation.

3.2 Formal Properties of IPT

In order to design a protocol solving the IPT problem, it is useful to consider the notion of immediate relevant predecessor of any event, whether relevant or not. First, we observe that, by definition, the immediate predecessor on P_j of an event e is necessarily the $lastr(e, j)$ event. Second, for $lastr(e, j)$ to be immediate predecessor of e, there must not be another $lastr(e, k)$ event on a path between $lastr(e, j)$ and e. These observations are formalized in the following definition:

DEFINITION 4. *Let $e \in H_i$. The set of immediate relevant predecessors of e (denoted $\mathcal{IP}(e)$), is the set of the relevant events $lastr(e, j)$ $(j = 1, \ldots, n)$ such that $\forall k : lastr(e, j) \not\in \uparrow (lastr(e, k))$.*

It follows from this definition that $\mathcal{IP}(e) \subseteq \{lastr(e, j) | j = 1, \ldots, n\} \subset \uparrow (e)$. When we consider Figure 1, The graph depicted in its right part describes the immediate predecessors of the relevant events of the computation defined in its left part, more precisely, a directed edge (e, f) means that the relevant event e is an immediate predecessor of the relevant event f ([3]).

The following lemmas show how the set of immediate predecessors of an event is related to those of its predecessors in the relation $\stackrel{hb}{\rightarrow}$. They will be used to design and prove the protocols solving the IPT problem. To ease the reading of the paper, their proofs are presented in Appendix A.

The intuitive meaning of the first lemma is the following: if e is not a receive event, all the causal paths arriving at e have $pred(e)$ as next-to-last event (see CP1). So, if $pred(e)$ is a relevant event, all the relevant events belonging to its relevant causal past are "separated" from e by $pred(e)$, and $pred(e)$ becomes the only immediate predecessor of e. In other words, the event $pred(e)$ constitutes a "reset" w.r.t. the set of immediate predecessors of e. On the other hand, if $pred(e)$ is not relevant, it does not separate its relevant causal past from e.

LEMMA 1. *If e is not a receive event, $\mathcal{IP}(e)$ is equal to:*
$\emptyset$ if $pred(e) = \perp$,
$\{pred(e)\}$ if $pred(e) \in R$,
$\mathcal{IP}(pred(e))$ if $pred(e) \not\in R$.

The intuitive meaning of the next lemma is as follows: if e is a receive event $receive(m)$, the causal paths arriving at e have either $pred(e)$ or $send(m)$ as next-to-last events. If $pred(e)$ is relevant, as explained in the previous lemma, this event "hides" from e all its relevant causal past and becomes an immediate predecessor of e. Concerning the last relevant predecessors of $send(m)$, only those that are not predecessors of $pred(e)$ remain immediate predecessors of e.

LEMMA 2. *Let $e \in H_i$ be the receive event of a message m. If $pred(e) \in R_i$, then, $\forall j$, $\mathcal{IP}(e) \cap R_j$ is equal to:*
$\{pred(e)\}$ if $j = i$,
$\emptyset$ if $lastr(pred(e), j) \geq lastr(send(m), j)$,
$\mathcal{IP}(send(m)) \cap R_j$ if $lastr(pred(e), j) < lastr(send(m), j)$.

The intuitive meaning of the next lemma is the following: if e is a receive event $receive(m)$, and $pred(e)$ is not relevant, the last relevant events in the relevant causal past of e are obtained by merging those of $pred(e)$ and those of $send(m)$ and by taking the latest on each process. So, the immediate predecessors of e are either those of $pred(e)$ or those of $send(m)$. On a process where the last relevant events of $pred(e)$ and of $send(m)$ are the same event f, none of the paths from f to e must contain another relevant event, and thus, f must be immediate predecessor of both events $pred(e)$ and $send(m)$.

LEMMA 3. *Let $e \in H_i$ be the receive event of a message m. If $pred(e) \not\in R_i$, then, $\forall j$, $\mathcal{IP}(e) \cap R_j$ is equal to:*
$\mathcal{IP}(pred(e)) \cap R_j$ if $lastr(pred(e), j) > lastr(send(m), j)$,
$\mathcal{IP}(send(m)) \cap R_j$ if $lastr(pred(e), j) < lastr(send(m), j)$
$\mathcal{IP}(pred(e)) \cap \mathcal{IP}(send(m)) \cap R_j$ if $lastr(pred(e), j) = lastr(send(m), j)$.

3.3 A Basic IPT Protocol

The basic protocol proposed here associates with each relevant event e, an attribute encoding the set $\mathcal{IP}(e)$ of its immediate predecessors. From the previous lemmas, the set

[3]Actually, this graph is the Hasse diagram of the partial order associated with the distributed computation.

$\mathcal{IP}(e)$ of any event e depends on the sets $\mathcal{IP}$ of the events $pred(e)$ and/or $send(m)$ (when $e = receive(m)$). Hence the idea to introduce a data structure allowing to manage the sets $\mathcal{IP}$s inductively on the poset $(H, \stackrel{hb}{\rightarrow})$. To take into account the information from $pred(e)$, each process manages a boolean array IP_i such that, $\forall e \in H_i$ the value of IP_i when e occurs (denoted $e.IP_i$) is the boolean array representation of the set $\mathcal{IP}(e)$. More precisely, $\forall j : IP_i[j] = 1 \Leftrightarrow lastr(e, j) \in \mathcal{IP}(e)$. As recalled in Section 2.3, the knowledge of $lastr(e, j)$ (for every e and every j) is based on the management of vectors VC_i. Thus, the set $\mathcal{IP}(e)$ is determined in the following way:

$$\mathcal{IP}(e) = \{e_j^y \mid e.VC_i[j] = y \wedge e.IP_i[j] = 1, j = 1, \ldots, n\}$$

Each process P_i updates IP_i according to the Lemmas 1, 2, and 3:

1. It results from Lemma 1 that, if e is not a receive event, the current value of IP_i is sufficient to determine $e.IP_i$. It results from Lemmas 2 and 3 that, if e is a receive event ($e = receive(m)$), then determining $e.IP_i$ involves information related to the event $send(m)$. More precisely, this information involves $\mathcal{IP}(send(m))$ and the timestamp of $send(m)$ (needed to compare the events $lastr(send(m), j)$ and $lastr(pred(e), j)$, for every j). So, both vectors $send(m).VC_j$ and $send(m).IP_j$ (assuming $send(m)$ produced by P_j) are attached to message m.
2. Moreover, IP_i must be updated upon the occurrence of each event. In fact, the value of IP_i just after an event e is used to determine the value $succ(e).IP_i$. In particular, as stated in the Lemmas, the determination of $succ(e).IP_i$ depends on whether e is relevant or not. Thus, the value of IP_i just after the occurrence of event e must " keep track" of this event.

The following protocol, previously presented in [4] without proof, ensures the correct management of arrays VC_i (as in Section 2.3) and IP_i (according to the Lemmas of Section 3.2). The timestamp associated with a relevant event e is denoted $e.TS$.

R0 Initialization: Both $VC_i[1..n]$ and $IP_i[1..n]$ are initialized to $[0, \ldots, 0]$.

R1 Each time it produces a relevant event e:

- P_i associates with e the timestamp $e.TS$ defined as follows $e.TS = \{(k, VC_i[k]) \mid IP_i[k] = 1\}$,
- P_i increments its vector clock entry $VC_i[i]$ (namely it executes $VC_i[i] := VC_i[i] + 1$),
- P_i resets IP_i: $\forall \ell \neq i : IP_i[\ell] := 0; IP_i[i] := 1$.

R2 When P_i sends a message m to P_j, it attaches to m the current values of VC_i (denoted $m.VC$) and the boolean array IP_i (denoted $m.IP$).

R3 When it receives a message m from P_j, P_i executes the following updates:

$\forall k \in [1..n]$: **case**
$VC_i[k] < m.VC[k]$ **then** $VC_i[k] := m.VC[k]$; $IP_i[k] := m.IP[k]$
$VC_i[k] = m.VC[k]$ **then** $IP_i[k] := \min(IP_i[k], m.IP[k])$
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

The proof of the following theorem directly follows from Lemmas 1, 2 and 3.

THEOREM 1. *The protocol described in Section 3.3 solves the IPT problem: for any relevant event e, the timestamp e.TS contains the identifiers of all its immediate predecessors and no other event identifier.*

4. A GENERAL CONDITION

This section addresses a previously open problem, namely, "How to solve the IPT problem without requiring each application message to piggyback a whole vector clock and a whole boolean array?". First, a general condition that characterizes which entries of vectors VC_i and IP_i can be omitted from the control information attached to a message sent in the computation, is defined (Section 4.1). It is then shown (Section 4.2) that this condition is both sufficient and necessary.

However, this general condition cannot be locally evaluated by a process that is about to send a message. Thus, locally evaluable approximations of this general condition must be defined. To each approximation corresponds a protocol, implemented with additional local data structures. In that sense, the general condition defines a family of IPT protocols, that solve the previously open problem. This issue is addressed in Section 5.

4.1 To Transmit or Not to Transmit Control Information

Let us consider the previous IPT protocol (Section 3.3). Rule R3 shows that a process P_j does not systematically update each entry $VC_j[k]$ each time it receives a message m from a process P_i: there is no update of $VC_j[k]$ when $VC_j[k] \geq m.VC[k]$. In such a case, the value $m.VC[k]$ is useless, and could be omitted from the control information transmitted with m by P_i to P_j.

Similarly, some entries $IP_j[k]$ are not updated when a message m from P_i is received by P_j. This occurs when $0 < VC_j[k] = m.VC[k] \ \wedge \ m.IP[k] = 1$, or when $VC_j[k] > m.VC[k]$, or when $m.VC[k] = 0$ (in the latest case, as $m.IP[k] = IP_i[k] = 0$ then no update of $IP_j[k]$ is necessary). Differently, some other entries are systematically reset to 0 (this occurs when $0 < VC_j[k] = m.VC[k] \wedge m.IP[k] = 0$).

These observations lead to the definition of the condition $K(m, k)$ that characterizes which entries of vectors VC_i and IP_i can be omitted from the control information attached to a message m sent by a process P_i to a process P_j:

DEFINITION 5. $K(m, k) \equiv$
$(send(m).VC_i[k] = 0)$
$\vee \ (send(m).VC_i[k] < pred(receive(m)).VC_j[k])$
$\vee \ \big((send(m).VC_i[k] = pred(receive(m)).VC_j[k])$
$\wedge (send(m).IP_i[k] = 1)\big)$.

4.2 A Necessary and Sufficient Condition

We show here that the condition $K(m, k)$ is both necessary and sufficient to decide which triples of the form $(k, send(m).VC_i[k], send(m).IP_i[k])$ can be omitted in an outgoing message m sent by P_i to P_j. A triple attached to m will also be denoted $(k, m.VC[k], m.IP[k])$. Due to space limitations, the proofs of Lemma 4 and Lemma 5 are given in [1]. (The proof of Theorem 2 follows directly from these lemmas.)

LEMMA 4. *(Sufficiency) If $K(m,k)$ is true, then the triple $(k, m.VC[k], m.IP[k])$ is useless with respect to the correct management of $IP_j[k]$ and $VC_j[k]$.*

LEMMA 5. *(Necessity) If $K(m,k)$ is false, then the triple $(k, m.VC[k], m.IP[k])$ is necessary to ensure the correct management of $IP_j[k]$ and $VC_j[k]$.*

THEOREM 2. *When a process P_i sends m to a process P_j, the condition $K(m,k)$ is both necessary and sufficient not to transmit the triple $(k, send(m).VC_i[k], send(m).IP_i[k])$.*

5. A FAMILY OF IPT PROTOCOLS BASED ON EVALUABLE CONDITIONS

It results from the previous theorem that, if P_i could evaluate $K(m,k)$ when it sends m to P_j, this would allow us improve the previous IPT protocol in the following way: in rule R2, the triple $(k, VC_i[k], IP_i[k])$ is transmitted with m only if $\neg K(m,k)$. Moreover, rule R3 is appropriately modified to consider only triples carried by m. However, as previously mentioned, P_i cannot locally evaluate $K(m,k)$ when it is about to send m. More precisely, when P_i sends m to P_j, P_i knows the exact values of $send(m).VC_i[k]$ and $send(m).IP_i[k]$ (they are the current values of $VC_i[k]$ and $IP_i[k]$). But, as far as the value of $pred(receive(m)).VC_j[k]$ is concerned, two cases are possible. Case (i): If $pred(receive(m)) \stackrel{hb}{\rightarrow} send(m)$, then P_i can know the value of $pred(receive(m)).VC_j[k]$ and consequently can evaluate $K(m,k)$. Case (ii): If $pred(receive(m))$ and $send(m)$ are concurrent, P_i cannot know the value of $pred(receive(m)).VC_j[k]$ and consequently cannot evaluate $K(m,k)$. Moreover, when it sends m to P_j, whatever the case (i or ii) that actually occurs, P_i has no way to know which case does occur. Hence the idea to define *evaluable* approximations of the general condition. Let $K'(m,k)$ be an approximation of $K(m,k)$, that can be evaluated by a process P_i when it sends a message m. To be correct, the condition K' must ensure that, every time P_i should transmit a triple $(k, VC_i[k], IP_i[k])$ according to Theorem 2 (i.e., each time $\neg K(m,k)$), then P_i transmits this triple when it uses condition K'. Hence, the definition of a correct evaluable approximation:

DEFINITION 6. *A condition K', locally evaluable by a process when it sends a message m to another process, is* correct *if $\forall (m,k) : \neg K(m,k) \Rightarrow \neg K'(m,k)$ or, equivalently, $\forall (m,k) : K'(m,k) \Rightarrow K(m,k)$.*

This definition means that a protocol evaluating K' to decide which triples must be attached to messages, does not miss triples whose transmission is required by Theorem 2.

Let us consider the "constant" condition (denoted $K1$), that is always false, i.e., $\forall (m,k) : K1(m,k) = false$. This trivially correct approximation of K actually corresponds to the particular IPT protocol described in Section 3 (in which each message carries a whole vector clock and a whole boolean vector). The next section presents a better approximation of K (denoted $K2$).

5.1 A Boolean Matrix-Based Evaluable Condition

Condition $K2$ is based on the observation that condition K is composed of sub-conditions. Some of them can be locally evaluated while the others cannot. More precisely, $K \equiv a \vee \alpha \vee (\beta \wedge b)$, where $a \equiv send(m).VC_i[k] = 0$ and $b \equiv send(m).IP_i[k] = 1$ are locally evaluable, whereas $\alpha \equiv send(m).VC_i[k] < pred(receive(m)).VC_j[k]$ and $\beta \equiv send(m).VC_i[k] = pred(receive(m)).VC_j[k]$ are not. But, from easy boolean calculus, $a \vee ((\alpha \vee \beta) \wedge b) \implies a \vee \alpha \vee (\beta \wedge b) \equiv K$. This leads to condition $K' \equiv a \vee (\gamma \wedge b)$, where $\gamma = \alpha \vee \beta \equiv send(m).VC_i[k] \leq pred(receive(m)).VC_j[k]$, i.e., $K' \equiv (send(m).VC_i[k] \leq pred(receive(m)).VC_j[k] \wedge send(m).IP_i[k] = 1) \vee send(m).VC_i[k] = 0$.

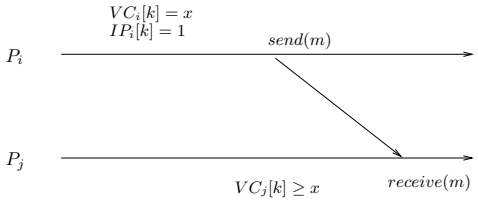


Figure 2: The Evaluable Condition $K2$

So, P_i needs to approximate the predicate $send(m).VC_i[k] \leq pred(receive(m)).VC_j[k]$. To be correct, this approximation has to be a locally evaluable predicate $c_i(j,k)$ such that, when P_i is about to send a message m to P_j, $c_i(j,k) \Rightarrow (send(m).VC_i[k] \leq pred(receive(m)).VC_j[k])$. Informally, that means that, when $c_i(j,k)$ holds, the local context of P_i allows to deduce that the receipt of m by P_j will not lead to $VC_j[k]$ update ("P_j knows as much as P_i about P_k"). Hence, the "concrete" condition $K2$ is the following: $K2 \equiv send(m).VC_i[k] = 0 \vee (c_i(j,k) \wedge send(m).IP_i[k] = 1)$.

Let us now examine the design of such a predicate (denoted c_i). First, the case $j = i$ can be ignored, since it is assumed (Section 2.1) that a process never sends a message to itself. Second, in the case $j = k$, the relation $send(m).VC_i[j] \leq pred(receive(m)).VC_j[j]$ is always true, because the receipt of m by P_j cannot update $VC_j[j]$. Thus, $\forall j \neq i : c_i(j,j)$ must be true. Now, let us consider the case where $j \neq i$ and $j \neq k$ (Figure 2). Suppose that there exists an event $e' = receive(m')$ with $e' < send(m)$, m' sent by P_j and piggybacking the triple $(k, m'.VC[k], m'.IP[k])$, and $m'.VC[k] \geq VC_i[k]$ (hence $m'.VC[k] = receive(m').VC_i[k]$). As $VC_j[k]$ cannot decrease this means that, as long as $VC_i[k]$ does not increase, for every message m sent by P_i to P_j we have the following: $send(m).VC_i[k] = receive(m').VC_i[k] = send(m').VC_j[k] \leq receive(m).VC_j[k]$, i.e., $c_i(j,k)$ must remain true. In other words, once $c_i(j,k)$ is true, the only event of P_i that could reset it to false is either the receipt of a message that increases $VC_i[k]$ or, if $k = i$, the occurrence of a relevant event (that increases $VC_i[i]$). Similarly, once $c_i(j,k)$ is false, the only event that can set it to true is the receipt of a message m' from P_j, piggybacking the triple $(k, m'.VC[k], m'.IP[k])$ with $m'.VC[k] \geq VC_i[k]$.

In order to implement the local predicates $c_i(j,k)$, each process P_i is equipped with a boolean matrix M_i (as in [11]) such that $M[j,k] = 1 \Leftrightarrow c_i(j,k)$. It follows from the previous discussion that this matrix is managed according to the following rules (note that its i-th line is not significant (case $j = i$), and that its diagonal is always equal to 1):

M0 Initialization: $\forall (j,k) : M_i[j,k]$ is initialized to 1.

M1 Each time it produces a relevant event e: P_i resets[4] the ith column of its matrix: $\forall j \neq i :\ M_i[j,i] := 0$.

M2 When P_i sends a message: no update of M_i occurs.

M3 When it receives a message m from P_j, P_i executes the following updates:

$\forall\ k \in [1..n]$: **case**
$VC_i[k] < m.VC[k]$ **then** $\forall \ell \neq i, j, k : M_i[\ell,k] := 0$; $M_i[j,k] := 1$
$VC_i[k] = m.VC[k]$ **then** $M_i[j,k] := 1$
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

The following lemma results from rules M0-M3. The theorem that follows shows that condition $K2(m,k)$ is correct. (Both are proved in [1].)

LEMMA 6. *$\forall i$, $\forall m$ sent by P_i to P_j, $\forall k$, we have:* $send(m).M_i[j,k] = 1 \Rightarrow send(m).VC_i[k] \leq pred(receive(m)).VC_j[k]$.

THEOREM 3. *Let m be a message sent by P_i to P_j. Let $K2(m,k) \equiv ((send(m).M_i[j,k] = 1) \wedge (send(m).IP_i[k] = 1) \vee (send(m).VC_i[k] = 0))$. We have: $K2(m,k) \Rightarrow K(m,k)$.*

5.2 Resulting IPT Protocol

The complete text of the IPT protocol based on the previous discussion follows.

RM0 Initialization:

- Both $VC_i[1..n]$ and $IP_i[1..n]$ are set to $[0, \ldots, 0]$, and $\forall\ (j,k) :\ M_i[j,k]$ is set to 1.

RM1 Each time it produces a relevant event e:

- P_i associates with e the timestamp $e.TS$ defined as follows: $e.TS = \{(k, VC_i[k]) \mid IP_i[k] = 1\}$,
- P_i increments its vector clock entry $VC_i[i]$ (namely, it executes $VC_i[i] := VC_i[i] + 1$),
- P_i resets IP_i: $\forall\ \ell \neq i :\ IP_i[\ell] := 0; IP_i[i] := 1$.
- P_i resets the ith column of its boolean matrix: $\forall j \neq i :\ M_i[j,i] := 0$.

RM2 When P_i sends a message m to P_j, it attaches to m the set of triples (each made up of a process id, an integer and a boolean): $\{(k, VC_i[k], IP_i[k]) \mid (M_i[j,k] = 0 \vee IP_i[k] = 0) \wedge (VC_i[k] > 0)\}$.

RM3 When P_i receives a message m from P_j, it executes the following updates:

$\forall$(k,$m.VC[k], m.IP[k]$) carried by m:
case
$VC_i[k] < m.VC[k]$ **then** $VC_i[k] := m.VC[k]$; $IP_i[k] := m.IP[k]$; $\forall \ell \neq i, j, k : M_i[\ell,k] := 0$; $M_i[j,k] := 1$
$VC_i[k] = m.VC[k]$ **then** $IP_i[k] := \min(IP_i[k], m.IP[k])$; $M_i[j,k] := 1$
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

5.3 A Tradeoff

The condition $K2(m,k)$ shows that a triple has not to be transmitted when $(M_i[j,k] = 1 \wedge IP_i[k] = 1) \vee (VC_i[k] > 0)$. Let us first observe that the management of $IP_i[k]$ is governed by the application program. More precisely, the IPT protocol does not define which are the relevant events, it has only to guarantee a correct management of $IP_i[k]$. Differently, the matrix M_i does not belong to the problem specification, it is an auxiliary variable of the IPT protocol, which manages it so as to satisfy the following implication when P_i sends m to P_j: $(M_i[j,k] = 1) \Rightarrow (pred(receive(m)).VC_j[k] \geq send(m).VC_i[k])$. The fact that the management of M_i is governed by the protocol and not by the application program leaves open the possibility to design a protocol where more entries of M_i are equal to 1. This can make the condition $K2(m,k)$ more often satisfied[5] and can consequently allow the protocol to transmit less triples.

We show here that it is possible to transmit less triples at the price of transmitting a few additional boolean vectors. The previous IPT matrix-based protocol (Section 5.2) is modified in the following way. The rules RM2 and RM3 are replaced with the modified rules RM2' and RM3' ($M_i[*,k]$ denotes the kth column of M_i).

RM2' When P_i sends a message m to P_j, it attaches to m the following set of 4-uples (each made up of a process id, an integer, a boolean and a boolean vector): $\{(k, VC_i[k], IP_i[k], M_i[*,k]) \mid (M_i[j,k] = 0 \vee IP_i[k] = 0) \wedge VC_i[k] > 0\}$.

RM3' When P_i receives a message m from P_j, it executes the following updates:

$\forall$(k,$m.VC[k], m.IP[k], m.M[1..n,k]$) carried by m:
case
$VC_i[k] < m.VC[k]$ **then** $VC_i[k] := m.VC[k]$; $IP_i[k] := m.IP[k]$; $\forall \ell \neq i :\ M_i[\ell,k] := m.M[\ell,k]$
$VC_i[k] = m.VC[k]$ **then** $IP_i[k] := \min(IP_i[k], m.IP[k])$; $\forall \ell \neq i :\ M_i[\ell,k] := max(M_i[\ell,k], m.M[\ell,k])$
$VC_i[k] > m.VC[k]$ **then** *skip*
endcase

Similarly to the proofs described in [1], it is possible to prove that the previous protocol still satisfies the property proved in Lemma 6, namely, $\forall i, \forall m$ sent by P_i to P_j, $\forall k$ we have $(send(m).M_i[j,k] = 1) \Rightarrow (send(m).VC_i[k] \leq pred(receive(m)).VC_j[k])$.

[4]Actually, the value of this column remains constant after its first update. In fact, $\forall j,\ M_i[j,i]$ can be set to 1 only upon the receipt of a message from P_j, carrying the value $VC_j[i]$ (see R3). But, as $M_j[i,i] = 1$, P_j does not send $VC_j[i]$ to P_i. So, it is possible to improve the protocol by executing this "reset" of the column $M_i[*,i]$ only when P_i produces its first relevant event.

[5]Let us consider the previously described protocol (Section 5.2) where the value of each matrix entry $M_i[j,k]$ is always equal to 0. The reader can easily verify that this setting correctly implements the matrix. Moreover, $K2(m,k)$ is then always false: it actually coincides with $K1(k,m)$ (which corresponds to the case where whole vectors have to be transmitted with each message).

Intuitively, the fact that some columns of matrices M are attached to application messages allows a *transitive* transmission of information. More precisely, the relevant history of P_k known by P_j is transmitted to a process P_i via a causal sequence of messages from P_j to P_i. In contrast, the protocol described in Section 5.2 used only a *direct* transmission of this information. In fact, as explained Section 5.1, the predicate c (locally implemented by the matrix M) was based on the existence of a message m' sent by P_j to P_i, piggybacking the triple $(k, m'.VC[k], m'.IP[k])$, and $m'.VC[k] \geq VC_i[k]$, i.e., on the existence of a *direct* transmission of information (by the message m').

The resulting IPT protocol (defined by the rules RM0, RM1, RM2' and RM3') uses the same condition $K2(m, k)$ as the previous one. It shows an interesting tradeoff between the number of triples $(k, VC_i[k], IP_i[k])$ whose transmission is saved and the number of boolean vectors that have to be additionally piggybacked. It is interesting to notice that the size of this additional information is bounded while each triple includes a non-bounded integer (namely a vector clock value).

6. EXPERIMENTAL STUDY

This section compares the behaviors of the previous protocols. This comparison is done with a simulation study. IPT1 denotes the protocol presented in Section 3.3 that uses the condition $K1(m, k)$ (which is always equal to *false*). IPT2 denotes the protocol presented in Section 5.2 that uses the condition $K2(m, k)$ where messages carry triples. Finally, IPT3 denotes the protocol presented in Section 5.3 that also uses the condition $K2(m, k)$ but where messages carry additional boolean vectors.

This section does not aim to provide an in-depth simulation study of the protocols, but rather presents a general view on the protocol behaviors. To this end, it compares IPT2 and IPT3 with regard to IPT1. More precisely, for IPT2 the aim was to evaluate the gain in terms of triples $(k, VC_i[k], IP_i[k])$ not transmitted with respect to the systematic transmission of whole vectors as done in IPT1. For IPT3, the aim was to evaluate the tradeoff between the additional boolean vectors transmitted and the number of saved triples. The behavior of each protocol was analyzed on a set of programs.

6.1 Simulation Parameters

The simulator provides different parameters enabling to tune both the communication and the processes features. These parameters allow to set the number of processes for the simulated computation, to vary the rate of communication (send/receive) events, and to alter the time duration between two consecutive relevant events. Moreover, to be independent of a particular topology of the underlying network, a fully connected network is assumed. Internal events have not been considered.

Since the presence of the triples $(k, VC_i[k], IP_i[k])$ piggybacked by a message strongly depends on the frequency at which relevant events are produced by a process, different time distributions between two consecutive relevant events have been implemented (e.g., normal, uniform, and Poisson distributions). The senders of messages are chosen according to a random law. To exhibit particular configurations of a distributed computation a given scenario can be provided to the simulator. Message transmission delays follow a standard normal distribution. Finally, the last parameter of the simulator is the number of send events that occurred during a simulation.

6.2 Parameter Settings

To compare the behavior of the three IPT protocols, we performed a large number of simulations using different parameters setting. We set to 10 the number of processes participating to a distributed computation. The number of communication events during the simulation has been set to 10 000. The parameter λ of the Poisson time distribution (λ is the average number of relevant events in a given time interval) has been set so that the relevant events are generated at the beginning of the simulation. With the uniform time distribution, a relevant event is generated (in the average) every 10 communication events. The location parameter of the standard normal time distribution has been set so that the occurrence of relevant events is shifted around the third part of the simulation experiment.

As noted previously, the simulator can be fed with a given scenario. This allows to analyze the worst case scenarios for IPT2 and IPT3. These scenarios correspond to the case where the relevant events are generated at the maximal frequency (i.e., each time a process sends or receives a message, it produces a relevant event).

Finally, the three IPT protocols are analyzed with the same simulation parameters.

6.3 Simulation Results

The results are displayed on the Figures 3.a-3.d. These figures plot the gain of the protocols in terms of the number of triples that are not transmitted (y axis) with respect to the number of communication events (x axis). From these figures, we observe that, whatever the time distribution followed by the relevant events, both IPT2 and IPT3 exhibit a behavior better than IPT1 (i.e., the total number of piggybacked triples is lower in IPT2 and IPT3 than in IPT1), even in the worst case (see Figure 3.d).

Let us consider the worst scenario. In that case, the gain is obtained at the very beginning of the simulation and lasts as long as it exists a process P_j for which $\forall k : \ VC_j[k] = 0$. In that case, the condition $\forall k : \ K(m, k)$ is satisfied. As soon as $\exists k : \ VC_j[k] \neq 0$, both IPT2 and IPT3 behave as IPT1 (the shape of the curve becomes flat) since the condition $K(m, k)$ is no longer satisfied.

Figure 3.a shows that during the first events of the simulation, the slope of curves IPT2 and IPT3 are steep. The same occurs in Figure 3.d (that depicts the worst case scenario). Then the slope of these curves decreases and remains constant until the end of the simulation. In fact, as soon as $VC_j[k]$ becomes greater than 0, the condition $\neg K(m, k)$ reduces to $(M_i[j, k] = 0 \vee IP_i[k] = 0)$.

Figure 3.b displays an interesting feature. It considers $\lambda = 100$. As the relevant events are taken only during the very beginning of the simulation, this figure exhibits a very steep slope as the other figures. The figure shows that, as soon as no more relevant events are taken, on average, 45% of the triples are not piggybacked by the messages. This shows the importance of matrix M_i. Furthermore, IPT3 benefits from transmitting additional boolean vectors to save triple transmissions. The Figures 3.a-3.c show that the average gain of IPT3 with respect to IPT2 is close to 10%.

Finally, Figure 3.c underlines even more the importance

of matrix M_i. When very few relevant events are taken, IPT2 and IPT3 turn out to be very efficient. Indeed, this figure shows that, very quickly, the gain in number of triples that are saved is very high (actually, 92% of the triples are saved).

6.4 Lessons Learned from the Simulation

Of course, all simulation results are consistent with the theoretical results. IPT3 is always better than or equal to IPT2, and IPT2 is always better than IPT1. The simulation results teach us more:

• The first lesson we have learnt concerns the matrix M_i. Its use is quite significant but mainly depends on the time distribution followed by the relevant events. On the one hand, when observing Figure 3.b where a large number of relevant events are taken in a very short time, IPT2 can save up to 45% of the triples. However, we could have expected a more sensitive gain of IPT2 since the boolean vector IP tends to stabilize to $[1, ..., 1]$ when no relevant events are taken. In fact, as discussed in Section 5.3, the management of matrix M_i within IPT2 does not allow a transitive transmission of information but only a direct transmission of this information. This explains why some columns of M_i may remain equal to 0 while they could potentially be equal to 1. Differently, as IPT3 benefits from transmitting additional boolean vectors (providing a transitive transmission information) it reaches a gain of 50%.
On the other hand, when very few relevant events are taken in a large period of time (see Figure 3.c), the behavior of IPT2 and IPT3 turns out to be very efficient since the transmission of up to 92% of the triples is saved. This comes from the fact that very quickly the boolean vector IP_i tends to stabilize to $[1, ..., 1]$ and that matrix M_i contains very few 0 since very few relevant events have been taken. Thus, a direct transmission of the information is sufficient to quickly get matrices M_i equal to $[1, ..., 1], \ldots, [1, ..., 1]$.

• The second lesson concerns IPT3, more precisely, the tradeoff between the additional piggybacking of boolean vectors and the number of triples whose transmission is saved. With $n = 10$, adding 10 booleans to a triple does not substantially increases its size. The Figures 3.a-3.c exhibit the number of triples whose transmission is saved: the average gain (in number of triples) of IPT3 with respect to IPT2 is about 10%.

7. CONCLUSION

This paper has addressed an important causality-related distributed computing problem, namely, the *Immediate Predecessors Tracking* problem. It has presented a family of protocols that provide each relevant event with a timestamp that exactly identify its immediate predecessors. The family is defined by a general condition that allows application messages to piggyback control information whose size can be smaller than n (the number of processes). In that sense, this family defines message size-efficient IPT protocols. According to the way the general condition is implemented, different IPT protocols can be obtained. Three of them have been described and analyzed with simulation experiments. Interestingly, it has also been shown that the efficiency of the protocols (measured in terms of the size of the control information that is not piggybacked by an application message) depends on the pattern defined by the communication events and the relevant events.

Last but not least, it is interesting to note that if one is not interested in tracking the immediate predecessor events, the protocols presented in the paper can be simplified by suppressing the IP_i booleans vectors (but keeping the boolean matrices M_i). The resulting protocols, that implement a vector clock system, are particularly efficient as far as the size of the timestamp carried by each message is concerned. Interestingly, this efficiency is not obtained at the price of additional assumptions (such as FIFO channels).

8. REFERENCES

[1] Anceaume E., Hélary J.-M. and Raynal M., Tracking Immediate Predecessors in Distributed Computations. *Res. Report #1344*, IRISA, Univ. Rennes (France), 2001.

[2] Baldoni R., Prakash R., Raynal M. and Singhal M., Efficient Δ-Causal Broadcasting. *Journal of Computer Systems Science and Engineering*, 13(5):263-270, 1998.

[3] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems*, 3(1):63-75, 1985.

[4] Diehl C., Jard C. and Rampon J.-X., Reachability Analysis of Distributed Executions, *Proc. TAPSOFT'93*, Springer-Verlag LNCS 668, pp. 629-643, 1993.

[5] Fidge C.J., Timestamps in Message-Passing Systems that Preserve Partial Ordering, *Proc. 11th Australian Computing Conference*, pp. 56-66, 1988.

[6] Fromentin E., Jard C., Jourdan G.-V. and Raynal M., On-the-fly Analysis of Distributed Computations, *IPL*, 54:267-274, 1995.

[7] Fromentin E. and Raynal M., Shared Global States in Distributed Computations, *JCSS*, 55(3):522-528, 1997.

[8] Fromentin E., Raynal M., Garg V.K. and Tomlinson A., On-the-Fly Testing of Regular Patterns in Distributed Computations. *Proc. ICPP'94*, Vol. 2:73-76, 1994.

[9] Garg V.K., *Principles of Distributed Systems*, Kluwer Academic Press, 274 pages, 1996.

[10] Hélary J.-M., Mostéfaoui A., Netzer R.H.B. and Raynal M., Communication-Based Prevention of Useless Ckeckpoints in Distributed Computations. *Distributed Computing*, 13(1):29-43, 2000.

[11] Hélary J.-M., Melideo G. and Raynal M., Tracking Causality in Distributed Systems: a Suite of Efficient Protocols. *Proc. SIROCCO'00*, Carleton University Press, pp. 181-195, L'Aquila (Italy), June 2000.

[12] Hélary J.-M., Netzer R. and Raynal M., Consistency Issues in Distributed Checkpoints. *IEEE TSE*, 25(4):274-281, 1999.

[13] Hurfin M., Mizuno M., Raynal M. and Singhal M., Efficient Distributed Detection of Conjunction of Local Predicates in Asynch Computations. *IEEE TSE*, 24(8):664-677, 1998.

[14] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558-565, 1978.

[15] Marzullo K. and Sabel L., Efficient Detection of a Class of Stable Properties. *Distributed Computing*, 8(2):81-91, 1994.

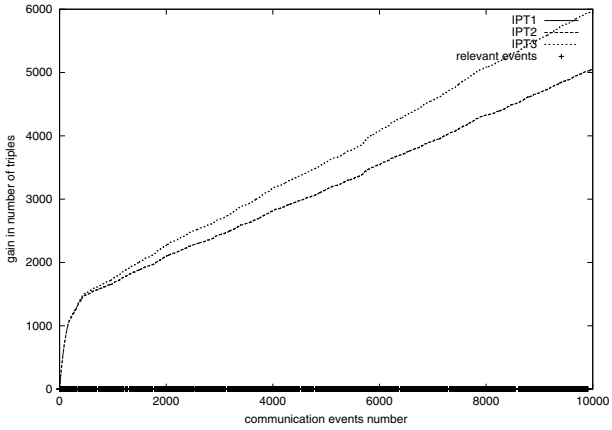
[16] Mattern F., Virtual Time and Global States of Distributed Systems. *Proc. Int. Conf. Parallel and Distributed Algorithms*, (Cosnard, Quinton, Raynal, Robert Eds), North-Holland, pp. 215-226, 1988.

[17] Prakash R., Raynal M. and Singhal M., An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environment. *JPDC*, 41:190-204, 1997.

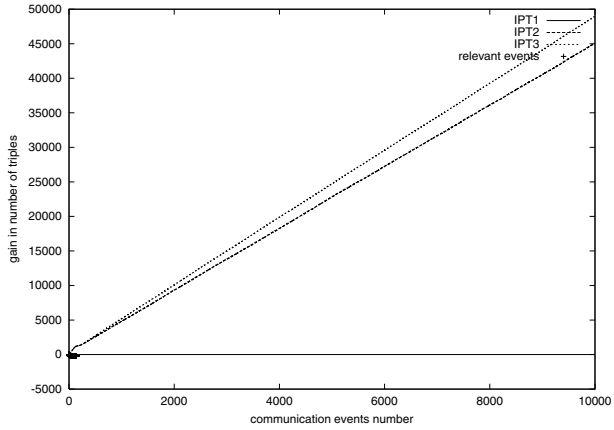
[18] Raynal M. and Singhal S., Logical Time: Capturing Causality in Distributed Systems. *IEEE Computer*, 29(2):49-57, 1996.

[19] Singhal M. and Kshemkalyani A., An Efficient Implementation of Vector Clocks. *IPL*, 43:47-52, 1992.

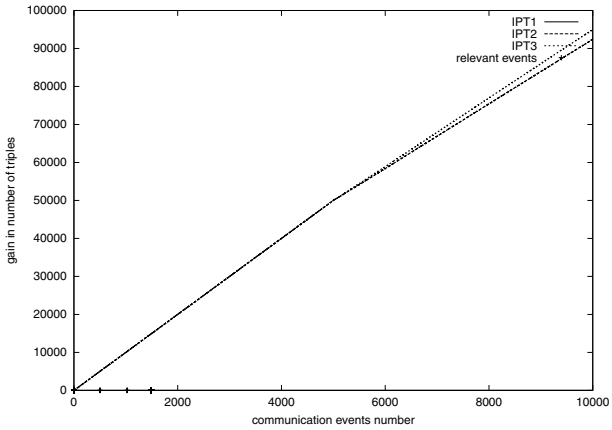
[20] Wang Y.M., Consistent Global Checkpoints That Contain a Given Set of Local Checkpoints. *IEEE TOC*, 46(4):456-468, 1997.



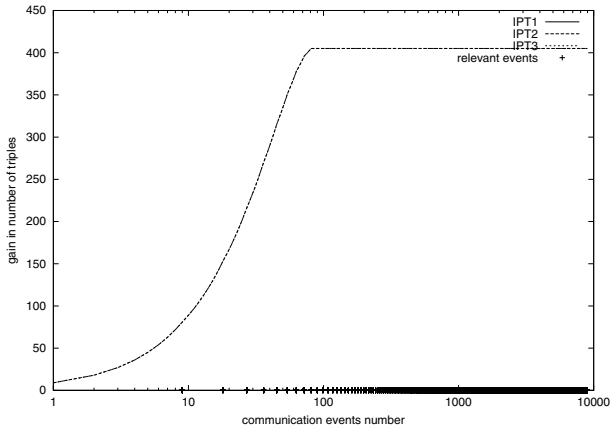
(a) The relevant events follow a uniform distribution (ratio=1/10)



(b) The relevant events follow a Poisson distribution ($\lambda = 100$)



(c) The relevant events follow a normal distribution



(d) For each p_i, p_i takes a relevant event and broadcast to all processes

Figure 3: Experimental Results